Spectrometric solutions from components to systems

JETI TECHNISCHE INSTRUMENTE GMBH

Technical Note 17

Controlling the specbos / spectraval devices using the JETI SDK

Introduction

JETI's **specbos 1201/1211**, **spectraval 1501/1511** and **specbos 2501** can be controlled directly to make possible an implementation into customer specific programs.
The possibilities to control the instrument are as follows:

- by DLLs (in SDK) or
- by firmware commands

It can be freely chosen between these possibilities, but one should note, that DLLs are the easiest way to control the device (without any disadvantages or limitations in comparison to others).
Generally, it is possible:

- to change any settings
- to control hardware features of the device (like shutter or laser pointer)
- to make measurements
- to get measurement results
- to make calculations (a number of radiometric, photometric and colorimetric calculations can be done directly by the device).

DLLs and a firmware command list are attached to the standard delivery of a JETI **spectraval/specbos** devices.

Please note that the documentations on DLLs and the firmware are provided with full description of all functions for completeness but concerning many of them it is hard to imagine an external application where they could be useful. For example, functions which give spectra in terms of pixels (of the CCD) are widely used in internal testing software of JETI, but they can hardly be of interest for an external user. So, it is recommended not to depart from examples and schemes that are given hereafter.

Remarks on programming for spectraval 1501/1511 and specbos 1201/1211/2501

Integration time adaption

All **specbos 1201/1211/2501** and **spectraval 1501/1511** can automatically adapt their integration time to the brightness of the light source. Although it is always possible to use fixed integration times, it is really hard to imagine a non-special radiometric application, where fixed integration times should be used. Fixed integration times can lead to over- and underexposure because one can never exactly know the properties of the light source in advance, and consequently make all further calculations senseless (expecially in the case of overexposure). So, it is strongly recommended **always to use the automatic adaption**.

Low speed of specbos 1201

A **specbos 1201** is approximately 10 times slower than a **specbos 1211/2501** or **spectraval 1501/1511**. For example, while **specbos 1211** needs 5 seconds to proceed a measurement of some light source, **specbos 1201** needs 50 seconds (such integration times would practically mean, that the light source is pretty dark, e.g. a black color on a TV-set).
In such applications like monitor calibration, where many measurements are required, these times can add up to huge time periods of many hours.
So, it can have a sense to narrow the range of possible integration times of **specbos 1201** (for monitor calibrations it is recommended to set the maximal integration time to 4 seconds). Although it does lead to underexposure and therefore to measurement inaccuracy, it can be often tolerated in practice because users rarely have an appropriately darkened room which allows really accurate measurements of dark light sources.

Synchronization
Some light sources (PWM driven LEDs, displays and TV-sets are among them) are modulated. It means that their brightness very quickly changes in time – the light source "flickers". For example, for a display with a frequency of 50 Hz the periodicity of its "flickering" (modulation period) is 20 ms. While integration times of a **specbos 1201** are nearly always much longer than that, integration times of a **specbos 1211/2501** or **spectraval 1501/1511** can be almost of the same order and its advantage turns to disadvantage: high speed of **1211/1501/1511/2501** leads to loss of reproducibility due to the fact, that not always an even number of flickering-periods fits in the integration time of the device.
To solve this problem, **specbos 1211/2501** and **spectraval 1501/1511** allows synchronization with a light source, i.e. to provide a guarantee that the integration time contains an integer number of modulation periods.

Making synchronization correctly and user-friendly at the same time may be the most difficult problem for a software engineer.
First of all it is necessary to distinguish between **specbos 1201** and **specbos 1211/2501 / spectraval 1501/1511** and to grey out all synchronization features in case of **specbos 1201** if they are implemented in user interface.
Secondly, it is possible to measure synchronization frequency only of a reasonably bright light source (for monitor calibration it would be a good idea to turn – or ask a user to turn the monitor to 80 % white) and in absense of parasitic light of other light sources like another monitor in the immediate neighbourhood or ceiling lights.
Thirdly, sometimes a measurement of the synchronization frequency is still impossible even if all the conditions are fulfilled. For example, some monitors lose their modulation if they are set to maximal brightness. That is quite normal and in such a case synchronization can be switched off.

Summing up, it is important to give the user of a custom software necessary information and carefully differentiate cases when an error-message or a warning must be shown or neither.

DLLs

There exist 5 different DLLs in the SDK - spectro, spectro_ex, radio, radio_ex and core. The DLLs of interest for radiometric applications are the radio-, the radio_ex- and the core-DLL. It is necessary to use at least version V4.0.0 to run the following examples:
The *radio DLL* offers the basic functions for radiometric measurements. The parameters are fixed to the following standard values:

- Wavelength range VIS (380 ... 780 nm)
- Wavelength step 5 nm
- Automatic adaption of integration time (only)
- So, it is easy to use, but does not allow any variation.

The *radio_ex DLL* has more flexibility. Here you can set the main parameters to fit your application:

- Wavelength range (350 ... 1000 nm or 250 … 1000 nm for the UV version). Note that such values as color coordinates, luminance/illuminance, CCT and many others are defined and calculated only on the basis of spectra from 380 to 780 nm. Normally it makes a sense to enlarge the wavelength range only if radiance/irradiance values or spectra as such are of interest.
- Wavelength step 1 or 5 nm
- Integration time: fixed 0.01 ... 60 000.0 ms or automatically adapted (prefered)
- Averaged measurement

The *core DLL* is a pure "translation" of the firmware commands into the DLL language. So, it allows full flexibility, but it is necessary to go more into details. Nevertheless, it contains some hardware control functions (e.g. laser on/off, device reset, and others), which are needed almost in every advanced application.

Choice

It is recommended to use *radio_ex DLL* for LED and monitor measurement and similar applications, added by some commands of the *core DLL* (especially the target switch on/ off and the synchronization of the measurement).

Preconditions

The basics for the application of a DLL are described in chapter 2 and 3 of each individual DLL manual.
Please pay attention to the following issues:

- The DLLs can be mixed in an application.
- Copy the necessary DLL(s) into the Windows System folder or into the working directory of the calling application
- Use the stdcall convention for calling.

Measuring procedure (chapter 2.1 of each DLL guide)

To get access to the functions you must copy the files jeti_radio_ex.dll and jeti_core.dll to the working directory of your application, or to the windows\system32 directory.
In general, the user initiates communication with the target device(s) by making a call to JETI_GetNumRadioEx. This call will return the number of target devices. This number is then used as a range when calling JETI_GetSerialRadioEx to build a list of device serial numbers.

To access a device, it must first be opened by a call to JETI_OpenRadioEx using an index determined from the call to JETI_GetNumRadioEx. The JETI_OpenRadioEx function will return a handle to the device that is used in all subsequent accesses. When I/O operations are complete, the device is closed by a call to JETI_CloseRadioEx.
In case of a fatal communication error (error code 0xFF) JETI_HardReset (from jeti_core.dll) could be used to reset the device and resume the communication.

Code example

Principle structure of a connect-routine:

```
LibError = JETI_GetNumXXXX(NumDevices)
       //The name of the function is JETI_GetNumDevices if core.dll is used,
       //JETI_GetNumRadio if radio.dll is used,
       //JETI_GetNumRadioEx if radio_ex.dll is used (and the same way further).
       //Now the NumDevices variable contains the number of connected JETI-devices

If LibError<>0 Then ...
       // treat error, possibly end the program. Do so after calling every
       // DLL-function.

If NumDevices < 1 Then ...
       //no device is connected; error message; end the routine.

If NumDevices > 1 Then ...
       //more than 1 device is connected; error message; end the routine
       //or
       //use device no. 0 (all devices are ennumerated from 0 to NumDevices-1)
       //
       //or
       //(that would be the most proper treatment, however the most complex)
       //fill a list in a loop with serials of all connected devices like
       //     FOR i=0 to NumDevices-1
       //     LibError = JETI_GetSerialXXXX(i, Serial1, Serial2)
       //           If LibError<>0 Then ... //treat error
       //     //store i and Serial1 in the list somehow
       //     NEXT i
       //and let the user choose by himself

LibError = JETI_OpenXXXX(device_number_i_chosen_from_list, Device)
       //or LibError = LibError = JETI_OpenXXXX(0, Device) if you prefer the second
       //simplified solution
If LibError <> 0 Then ...
       //cannot connect; treat error
```

Comment: JETI_GetSerialXXXX and JETI_OpenXXXX are the only functions that work with internal device number as argument. After calling of JETI_OpenXXXX a handler (Device-variable) is initialized and it must be used for all actions later.

Principle structure of a parameter reading routine:

The purpose is to determine the device type and its performance, e.g. wavelength range, availability of synchronization etc.

```
//--------------- Determine type of the device -----------
LibError = JETI_GetDeviceType(Device, DevType)
       If LibError <> 0 Then ... //treat error

If DevType = 5 Then
//this is a specbos 2501
ELSE If DevType = 3 Then
//this is a spectraval 1501/1511
ELSE If DevType = 2 Then
//this is a specbos 1211
ELSE If DevType = 1 Then
//this is a specbos 1201
ELSE
//this is not spectraval 1501/1511 or specbos 2501/1211/1201
End If
```

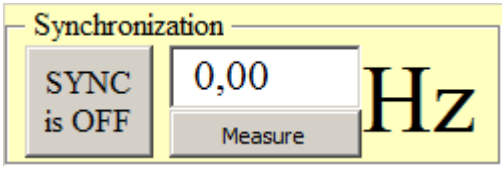
Principle structure of a laser switching routine:

The following scheme can be directly used for example if there is a button in the user interface and every click on it inverts the laser state. If the laser state is to be set explicitly, just use JETI_SetLaserStat with 1 or 0 as Target-argument to switch the laser on or off correspondingly.

```
Sub Button_Target_Click()
//define Target variable; Device is a global handler.

LibError = JETI_GetLaserStat(Device, Target)
       If LibError <> 0 Then ... //treat error

If Target = 0 Then Target = 1 Else Target = 0 //invert the value

LibError = JETI_SetLaserStat(Device, Target)
       If LibError <> 0 Then ... //treat error

If Target = 1 Then
//print somewhere that laser is on, or make the button red or something like that
Else
//print somewhere that laser is off, or make the button grey or something like that
End If

End Sub
```

Principle structure of a synchronization-routine:

Generally, the functionality of the synchronization consists of two parts: measuring (or setting manually) of the modulation frequency of a light source and then sending it to the device. So, the most complete interface operating with synchronization could look like following:



Assume we have a toggle button SyncToggle whose value can be TRUE or FALSE (for pressed and released states correspondingly), a button for starting measurement of the modulation frequency SyncMeasure, and a text-field storing a float value SyncFreq, which contains the measured or entered synchronization frequency in a range from 16 to 5000 Hz.

```
//---------------- Measure synchronization frequency ---------------
Sub SyncMeasure_Click()
//If the software controls a light source (e.g. monitor) directly, turn it bright. //If not -
make the user know (at least in the Help) that he should do it by //himself.

LibError = JETI_GetFlickerFreq(Device, FlickerFreq, Warning)
//FlickerFreq = measured modulation frequency in Hz

       If LibError <> 0 Then ... //treat error
       Select Case Warning
              Case 12
                     //error too fuzzy
              Case 11
                     //warning no modulation
              Case Else
                     //measurement successful
       End Select

//display FlickerFreq in the textbox if it's not zero,
//disable synchronisation otherwise
If FlickerFreq <>0 Then SyncFreq = FlickerFreq
End Sub


//---------------- Set synchronization frequency ---------------
Sub SetSyncMode
//assume SyncFreq contains the frequency (measured or entered) to be set
If SyncFreq <> 0 And SyncToggle = True Then   //if something is measured or set
       LibError = JETI_SetSyncMode(Device, 1)  //turn synchronization on
              If LibError <> 0 Then ... //treat error

       LibError = JETI_SetSyncFreq(Device, SyncFreq)
              If LibError <> 0 Then ... //treat error
Else                                          //if nothing is measured or set
       LibError = JETI_SetSyncMode(Device, 0)  //turn synchronization off
              If LibError <> 0 Then ... //treat error
End If
```

```
End Sub
```

Principle structure of a measuring routine using radio_ex.dll:

```
Call SetSyncMode
//if we can deal with a modulated light source, call the routine described above //for setting
of the synchronization mode; if we are sure, that the light source is //not modulated or we
have definitely a specbos 1201 – we don't need it

//print somewhere something like: "Performing measurement..."

LibError = JETI_MeasureEx(Device, 0.0, Averages, Step)
       If LibError <> 0 Then ... //treat error

Do
       LibError = JETI_MeasureStatusEx(Device, busy) //wait for finish
              If LibError <> 0 Then ... //treat error
Loop Until busy = 0

//Get chromaticity coordinates x and y
LibError = JETI_ChromxyEx(Device, Chromx, Chromy)
       If LibError <> 0 Then ... //treat error

LibError = JETI_SpecRadEx(Device, Start_WaveLength, End_WaveLength, *Sprad)
//fetches the entire spectrum from Beg_limit to Toend_limit in radiometric units; //which
exactly units they are, depends on the actual measuring head, which is //automatically detected
//ReferenceSpectrum[0]=counts for Start_WaveLength
//ReferenceSpectrum[1]=counts for Start_WaveLength+1
//etc.
       If LibError <> 0 Then ... //treat error

LibError = JETI_RadioEx(Device, Start_WaveLength, End_WaveLength, Radio)
//gets an integral radiometric value: radiance, irradiance, radiant flux etc. //depending on
current measuring head
       If LibError <> 0 Then ... //treat error

LibError = JETI_PhotoEx(Device, Photo)
//gets an integral photometric value: luminance, illuminance, luminous flux etc. //depending on
current measuring head
       If LibError <> 0 Then ... //treat error

//etc.

//got everything; print somewhere something like: "Device is ready"
```

Principle structure of a disconnect-routine:

```
LibError = JETI_CloseXXXX(Device)

       If LibError<>0 Then ... //treat error
```

```
//dont't forget to grey out everything that cannot work while the device is //disconnected
```

Additional. Making it possible to cancel a measurement

Assume that after starting a measurement we want to be able to interrupt it by clicking some button BreakButton. Here is an easy way to do this. What we need is to catch a click-event on the button and then modify our waiting-loop.

```
//--------- Click event on the button -----------------
Sub BreakButton _Click()
boolStopCommand = True     //This is some global variable.
End Sub

//---------- Changes on the waiting passage ----------
Do
       DoEvents      //Let the system to treat events in this loop.
       If boolStopCommand = True Then
              LibError = JETI_MeasureBreakEx(Device)
                     If LibError<>0 Then ... //treat error
              boolStopCommand = False
              //goto printing something like “Device is ready” missing all
              //data-fetching
       End If

             LibError = JETI_MeasureStatusEx(Device, busy)
              If LibError<>0 Then ... //treat error
Loop Until busy = 0 //wait for finish
```

For examples in C see the folder samples/c in JETI SDK (RadioSample and SyncSample).

Last modified: April 2025